Chapter 2

The Entity-Relationship Data Model

The process of designing a database begins with an analysis of what information the database must hold and what are the relationships among components of that information. Often, the structure of the database, called the *database schema*, is specified in one of several languages or notations suitable for expressing designs. After due consideration, the design is committed to a form in which it can be input to a DBMS, and the database takes on physical existence.

In this book, we shall use several design notations. We begin in this chapter with a traditional and popular approach called the "entity-relationship" (E/R) model. This model is graphical in nature, with boxes and arrows representing the essential data elements and their connections.

In Chapter 3 we turn our attention to the relational model, where the world is represented by a collection of tables. The relational model is somewhat restricted in the structures it can represent. However, the model is extremely simple and useful, and it is the model on which the major commercial DBMS's depend today. Often, database designers begin by developing a schema using the E/R or an object-based model, then translate the schema to the relational model for implementation.

Other models are covered in Chapter 4. In Section 4.2, we shall introduce ODL (Object Definition Language), the standard for object-oriented databases. Next, we see how object-oriented ideas have affected relational DBMS's, yielding a model often called "object-relational."

Section 4.6 introduces another modeling approach, called "semistructured data." This model has an unusual amount of flexibility in the structures that the data may form. We also discuss, in Section 4.7, the XML standard for modeling data as a hierarchically structured document, using "tags" (like HTML tags) to indicate the role played by text elements. XML is an important embodiment of the semistructured data model.

Figure 2.1 suggests how the E/R model is used in database design. We

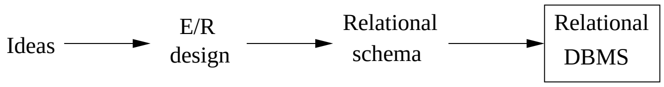

Figure 2.1: The database modeling and implementation process

start with ideas about the information we want to model and render them in the E/R model. The abstract E/R design is then converted to a schema in the data-specification language of some DBMS. Most commonly, this DBMS uses the relational model. If so, then by a fairly mechanical process that we shall discuss in Section 3.2, the abstract design is converted to a concrete, relational design, called a "relational database schema."

It is worth noting that, while DBMS's sometimes use a model other than relational or object-relational, there are no DBMS's that use the E/R model directly. The reason is that this model is not a sufficiently good match for the efficient data structures that must underlie the database.

2.1 Elements of the E/R Model

The most common model for abstract representation of the structure of a database is the *entity-relationship model* (or *E/R model*). In the E/R model, the structure of data is represented graphically, as an "entity-relationship diagram," using three principal element types:

1. Entity sets,
2. Attributes, and
3. Relationships.

We shall cover each in turn.

2.1.1 Entity Sets

An *entity* is an abstract object of some sort, and a collection of similar entities forms an *entity set*. There is some similarity between the entity and an "object" in the sense of object-oriented programming. Likewise, an entity set bears some resemblance to a class of objects. However, the E/R model is a static concept, involving the structure of data and not the operations on data. Thus, one would not expect to find methods associated with an entity set as one would with a class.

Example 2.1: We shall use as a running example a database about movies, their stars, the studios that produce them, and other aspects of movies. Each movie is an entity, and the set of all movies constitutes an entity set. Likewise, the stars are entities, and the set of stars is an entity set. A studio is another

E/R Model Variations

In some versions of the E/R model, the type of an attribute can be either:

1. Atomic, as in the version presented here.
2. A "struct," as in C, or tuple with a fixed number of atomic components.
3. A set of values of one type: either atomic or a "struct" type.

For example, the type of an attribute in such a model could be a set of pairs, each pair consisting of an integer and a string.

kind of entity, and the set of studios is a third entity set that will appear in our examples. □

2.1.2 Attributes

Entity sets have associated *attributes*, which are properties of the entities in that set. For instance, the entity set *Movies* might be given attributes such as *title* (the name of the movie) or *length*, the number of minutes the movie runs. In our version of the E/R model, we shall assume that attributes are atomic values, such as strings, integers, or reals. There are other variations of this model in which attributes can have some limited structure; see the box on "E/R Model Variations."

2.1.3 Relationships

Relationships are connections among two or more entity sets. For instance, if *Movies* and *Stars* are two entity sets, we could have a relationship *Stars-in* that connects movies and stars. The intent is that a movie entity m is related to a star entity s by the relationship *Stars-in* if s appears in movie m. While binary relationships, those between two entity sets, are by far the most common type of relationship, the E/R model allows relationships to involve any number of entity sets. We shall defer discussion of these multiway relationships until Section 2.1.7.

2.1.4 Entity-Relationship Diagrams

An *E/R diagram* is a graph representing entity sets, attributes, and relationships. Elements of each of these kinds are represented by nodes of the graph, and we use a special shape of node to indicate the kind, as follows:

- Entity sets are represented by rectangles.
- Attributes are represented by ovals.
- Relationships are represented by diamonds.

Edges connect an entity set to its attributes and also connect a relationship to its entity sets.

Example 2.2: In Fig. 2.2 is an E/R diagram that represents a simple database about movies. The entity sets are *Movies*, *Stars*, and *Studios*.

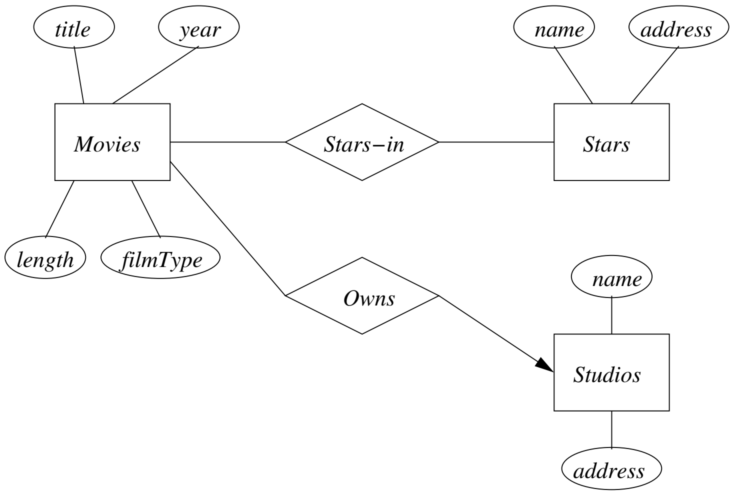

Figure 2.2: An entity-relationship diagram for the movie database

The *Movies* entity set has four attributes: *title*, *year* (in which the movie was made), *length*, and *filmType* (either "color" or "blackAndWhite"). The other two entity sets *Stars* and *Studios* happen to have the same two attributes: *name* and *address*, each with an obvious meaning. We also see two relationships in the diagram:

1. *Stars-in* is a relationship connecting each movie to the stars of that movie. This relationship consequently also connects stars to the movies in which they appeared.
2. *Owns* connects each movie to the studio that owns the movie. The arrow pointing to entity set *Studios* in Fig. 2.2 indicates that each movie is owned by a unique studio. We shall discuss uniqueness constraints such as this one in Section 2.1.6.

□

2.1.5 Instances of an E/R Diagram

E/R diagrams are a notation for describing the *schema* of databases, that is, their structure. A database described by an E/R diagram will contain particular data, which we call the database *instance*. Specifically, for each entity set, the database instance will have a particular finite set of entities. Each of these entities has particular values for each attribute. Remember, this data is abstract only; we do not store E/R data directly in a database. Rather, imagining this data exists helps us to think about our design, before we convert to relations and the data takes on physical existence.

The database instance also includes specific choices for the relationships of the diagram. A relationship R that connects n entity sets $E_1, E_2, \ldots, E_n$ has an instance that consists of a finite set of lists $(e_1, e_2, \ldots, e_n)$, where each e_i is chosen from the entities that are in the current instance of entity set E_i. We regard each of these lists of n entities as "connected" by relationship R.

This set of lists is called the *relationship set* for the current instance of R. It is often helpful to visualize a relationship set as a table. The columns of the table are headed by the names of the entity sets involved in the relationship, and each list of connected entities occupies one row of the table.

Example 2.3: An instance of the *Stars-in* relationship could be visualized as a table with pairs such as:

Movies	*Stars*
`Basic Instinct`	`Sharon Stone`
`Total Recall`	`Arnold Schwarzenegger`
`Total Recall`	`Sharon Stone`

The members of the relationship set are the rows of the table. For instance,

(Basic Instinct, Sharon Stone)

is a tuple in the relationship set for the current instance of relationship *Stars-in*. □

2.1.6 Multiplicity of Binary E/R Relationships

In general, a binary relationship can connect any member of one of its entity sets to any number of members of the other entity set. However, it is common for there to be a restriction on the "multiplicity" of a relationship. Suppose R is a relationship connecting entity sets E and F. Then:

- If each member of E can be connected by R to at most one member of F, then we say that R is *many-one* from E to F. Note that in a many-one relationship from E to F, each entity in F can be connected to many members of E. Similarly, if instead a member of F can be connected by R to at most one member of E, then we say R is many-one from F to E (or equivalently, one-many from E to F).

- If R is both many-one from E to F and many-one from F to E, then we say that R is *one-one*. In a one-one relationship an entity of either entity set can be connected to at most one entity of the other set.
- If R is neither many-one from E to F or from F to E, then we say R is *many-many*.

As we mentioned in Example 2.2, arrows can be used to indicate the multiplicity of a relationship in an E/R diagram. If a relationship is many-one from entity set E to entity set F, then we place an arrow entering F. The arrow indicates that each entity in set E is related to at most one entity in set F. Unless there is also an arrow on the edge to E, an entity in F may be related to many entities in E.

Example 2.4: Following this principle, a one-one relationship between entity sets E and F is represented by arrows pointing to both E and F. For instance, Fig. 2.3 shows two entity sets, *Studios* and *Presidents*, and the relationship *Runs* between them (attributes are omitted). We assume that a president can run only one studio and a studio has only one president, so this relationship is one-one, as indicated by the two arrows, one entering each entity set.

Figure 2.3: A one-one relationship

Remember that the arrow means "at most one"; it does not guarantee existence of an entity of the set pointed to. Thus, in Fig. 2.3, we would expect that a "president" is surely associated with some studio; how could they be a "president" otherwise? However, a studio might not have a president at some particular time, so the arrow from *Runs* to *Presidents* truly means "at most one" and not "exactly one." We shall discuss the distinction further in Section 2.3.6. □

2.1.7 Multiway Relationships

The E/R model makes it convenient to define relationships involving more than two entity sets. In practice, ternary (three-way) or higher-degree relationships are rare, but they are occasionally necessary to reflect the true state of affairs. A multiway relationship in an E/R diagram is represented by lines from the relationship diamond to each of the involved entity sets.

Example 2.5: In Fig. 2.4 is a relationship *Contracts* that involves a studio, a star, and a movie. This relationship represents that a studio has contracted with a particular star to act in a particular movie. In general, the value of an E/R relationship can be thought of as a relationship set of tuples whose

Implications Among Relationship Types

We should be aware that a many-one relationship is a special case of a many-many relationship, and a one-one relationship is a special case of a many-one relationship. That is, any useful property of many-many relationships applies to many-one relationships as well, and a useful property of many-one relationships holds for one-one relationships too. For example, a data structure for representing many-one relationships will work for one-one relationships, although it might not work for many-many relationships.

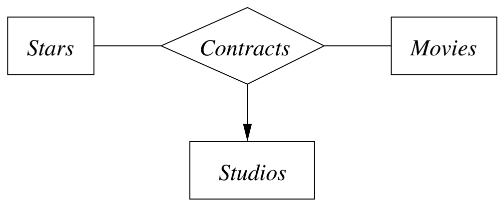

Figure 2.4: A three-way relationship

components are the entities participating in the relationship, as we discussed in Section 2.1.5. Thus, relationship *Contracts* can be described by triples of the form

$$(\text{studio, star, movie})$$

In multiway relationships, an arrow pointing to an entity set E means that if we select one entity from each of the other entity sets in the relationship, those entities are related to at most one entity in E. (Note that this rule generalizes the notation used for many-one, binary relationships.) In Fig. 2.4 we have an arrow pointing to entity set *Studios*, indicating that for a particular star and movie, there is only one studio with which the star has contracted for that movie. However, there are no arrows pointing to entity sets *Stars* or *Movies*. A studio may contract with several stars for a movie, and a star may contract with one studio for more than one movie. □

2.1.8 Roles in Relationships

It is possible that one entity set appears two or more times in a single relationship. If so, we draw as many lines from the relationship to the entity set as the entity set appears in the relationship. Each line to the entity set represents a different *role* that the entity set plays in the relationship. We therefore label the edges between the entity set and relationship by names, which we call "roles."

Limits on Arrow Notation in Multiway Relationships

There are not enough choices of arrow or no-arrow on the lines attached to a relationship with three or more participants. Thus, we cannot describe every possible situation with arrows. For instance, in Fig. 2.4, the studio is really a function of the movie alone, not the star and movie jointly, since only one studio produces a movie. However, our notation does not distinguish this situation from the case of a three-way relationship where the entity set pointed to by the arrow is truly a function of both other entity sets. In Section 3.4 we shall take up a formal notation — functional dependencies — that has the capability to describe all possibilities regarding how one entity set can be determined uniquely by others.

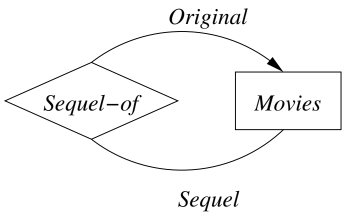

Figure 2.5: A relationship with roles

Example 2.6: In Fig. 2.5 is a relationship *Sequel-of* between the entity set *Movies* and itself. Each relationship is between two movies, one of which is the sequel of the other. To differentiate the two movies in a relationship, one line is labeled by the role *Original* and one by the role *Sequel*, indicating the original movie and its sequel, respectively. We assume that a movie may have many sequels, but for each sequel there is only one original movie. Thus, the relationship is many-one from *Sequel* movies to *Original* movies, as indicated by the arrow in the E/R diagram of Fig. 2.5. □

Example 2.7: As a final example that includes both a multiway relationship and an entity set with multiple roles, in Fig. 2.6 is a more complex version of the *Contracts* relationship introduced earlier in Example 2.5. Now, relationship *Contracts* involves two studios, a star, and a movie. The intent is that one studio, having a certain star under contract (in general, not for a particular movie), may further contract with a second studio to allow that star to act in a particular movie. Thus, the relationship is described by 4-tuples of the form

(studio1, studio2, star, movie),

meaning that studio2 contracts with studio1 for the use of studio1's star by studio2 for the movie.

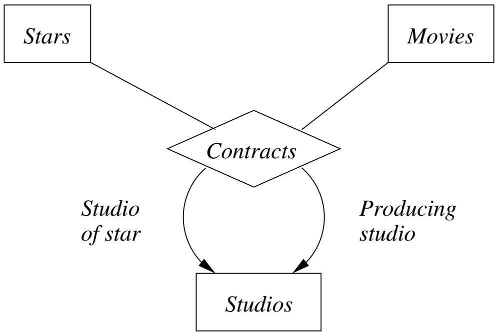

Figure 2.6: A four-way relationship

We see in Fig. 2.6 arrows pointing to *Studios* in both of its roles, as "owner" of the star and as producer of the movie. However, there are not arrows pointing to *Stars* or *Movies*. The rationale is as follows. Given a star, a movie, and a studio producing the movie, there can be only one studio that "owns" the star. (We assume a star is under contract to exactly one studio.) Similarly, only one studio produces a given movie, so given a star, a movie, and the star's studio, we can determine a unique producing studio. Note that in both cases we actually needed only one of the other entities to determine the unique entity—for example, we need only know the movie to determine the unique producing studio—but this fact does not change the multiplicity specification for the multiway relationship.

There are no arrows pointing to *Stars* or *Movies*. Given a star, the star's studio, and a producing studio, there could be several different contracts allowing the star to act in several movies. Thus, the other three components in a relationship 4-tuple do not necessarily determine a unique movie. Similarly, a producing studio might contract with some other studio to use more than one of their stars in one movie. Thus, a star is not determined by the three other components of the relationship. □

2.1.9 Attributes on Relationships

Sometimes it is convenient, or even essential, to associate attributes with a relationship, rather than with any one of the entity sets that the relationship connects. For example, consider the relationship of Fig. 2.4, which represents contracts between a star and studio for a movie.[1] We might wish to record the salary associated with this contract. However, we cannot associate it with the star; a star might get different salaries for different movies. Similarly, it does not make sense to associate the salary with a studio (they may pay different

[1] Here, we have reverted to the earlier notion of three-way contracts in Example 2.5, not the four-way relationship of Example 2.7.

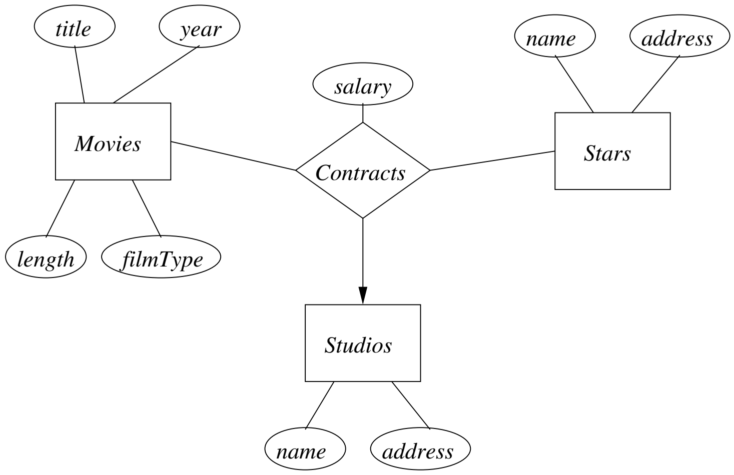

Figure 2.7: A relationship with an attribute

salaries to different stars) or with a movie (different stars in a movie may receive different salaries).

However, it is appropriate to associate a salary with the

(star, movie, studio)

triple in the relationship set for the *Contracts* relationship. In Fig. 2.7 we see Fig. 2.4 fleshed out with attributes. The relationship has attribute *salary*, while the entity sets have the same attributes that we showed for them in Fig. 2.2.

It is never necessary to place attributes on relationships. We can instead invent a new entity set, whose entities have the attributes ascribed to the relationship. If we then include this entity set in the relationship, we can omit the attributes on the relationship itself. However, attributes on a relationship are a useful convention, which we shall continue to use where appropriate.

Example 2.8: Let us revise the E/R diagram of Fig. 2.7, which has the salary attribute on the *Contracts* relationship. Instead, we create an entity set *Salaries*, with attribute *salary*. *Salaries* becomes the fourth entity set of relationship *Contracts*. The whole diagram is shown in Fig. 2.8. □

2.1.10 Converting Multiway Relationships to Binary

There are some data models, such as ODL (Object Definition Language), which we introduce in Section 4.2, that limit relationships to be binary. Thus, while the E/R model does not require binary relationships, it is useful to observe that any relationship connecting more than two entity sets can be converted to a collection of binary, many-one relationships. We can introduce a new entity set

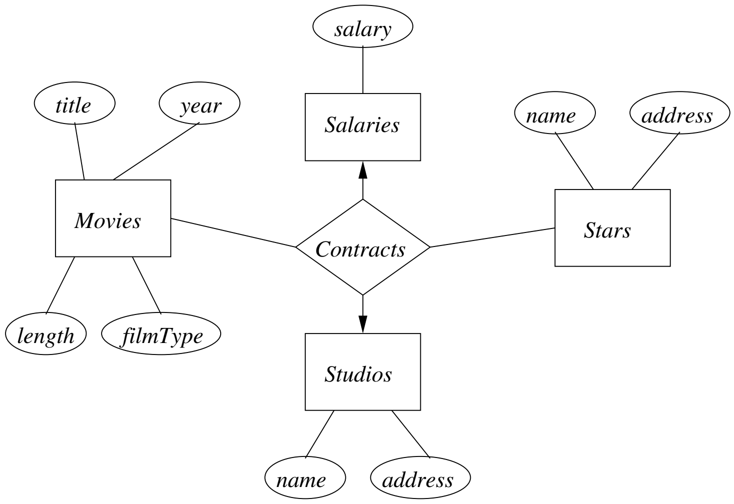

Figure 2.8: Moving the attribute to an entity set

whose entities we may think of as tuples of the relationship set for the multiway relationship. We call this entity set a *connecting* entity set. We then introduce many-one relationships from the connecting entity set to each of the entity sets that provide components of tuples in the original, multiway relationship. If an entity set plays more than one role, then it is the target of one relationship for each role.

Example 2.9: The four-way *Contracts* relationship in Fig. 2.6 can be replaced by an entity set that we may also call *Contracts*. As seen in Fig. 2.9, it participates in four relationships. If the relationship set for the relationship *Contracts* has a 4-tuple

(studio1, studio2, star, movie)

then the entity set *Contracts* has an entity e. This entity is linked by relationship *Star-of* to the entity *star* in entity set *Stars*. It is linked by relationship *Movie-of* to the entity *movie* in *Movies*. It is linked to entities *studio1* and *studio2* of *Studios* by relationships *Studio-of-star* and *Producing-studio*, respectively.

Note that we have assumed there are no attributes of entity set *Contracts*, although the other entity sets in Fig. 2.9 have unseen attributes. However, it is possible to add attributes, such as the date of signing, to entity set *Contracts*. □

2.1.11 Subclasses in the E/R Model

Often, an entity set contains certain entities that have special properties not associated with all members of the set. If so, we find it useful to define certain

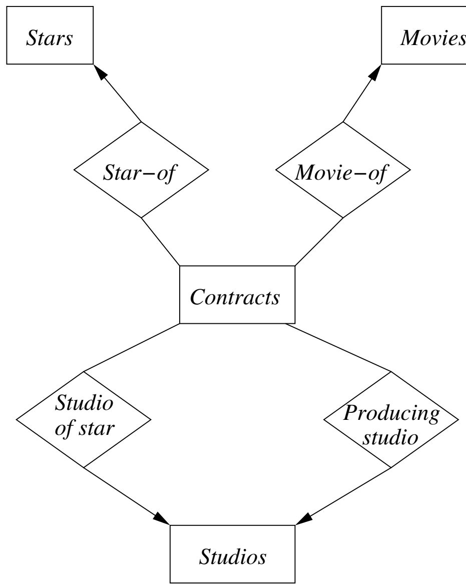

Figure 2.9: Replacing a multiway relationship by an entity set and binary relationships

special-case entity sets, or *subclasses*, each with its own special attributes and/or relationships. We connect an entity set to its subclasses using a relationship called *isa* (i.e., "an A is a B" expresses an "isa" relationship from entity set A to entity set B).

An isa relationship is a special kind of relationship, and to emphasize that it is unlike other relationships, we use for it a special notation. Each isa relationship is represented by a triangle. One side of the triangle is attached to the subclass, and the opposite point is connected to the superclass. Every isa relationship is one-one, although we shall not draw the two arrows that are associated with other one-one relationships.

Example 2.10: Among the kinds of movies we might store in our example database are cartoons, murder mysteries, adventures, comedies, and many other special types of movies. For each of these movie types, we could define a subclass of the entity set *Movies*. For instance, let us postulate two subclasses: *Cartoons* and *Murder-Mysteries*. A cartoon has, in addition to the attributes and relationships of *Movies* an additional relationship called *Voices* that gives us a set of stars who speak, but do not appear in the movie. Movies that are not cartoons do not have such stars. Murder-mysteries have an additional attribute *weapon*. The connections among the three entity sets *Movies*, *Cartoons*, and *Murder-Mysteries* is shown in Fig. 2.10. □

While, in principle, a collection of entity sets connected by *isa* relationships

Parallel Relationships Can Be Different

Figure 2.9 illustrates a subtle point about relationships. There are two different relationships, *Studio-of-Star* and *Producing-Studio*, that each connect entity sets *Contracts* and *Studios*. We should not presume that these relationships therefore have the same relationship sets. In fact, in this case, it is unlikely that both relationships would ever relate the same contract to the same studios, since a studio would then be contracting with itself.

More generally, there is nothing wrong with an E/R diagram having several relationships that connect the same entity sets. In the database, the instances of these relationships will normally be different, reflecting the different meanings of the relationships. In fact, if the relationship sets for two relationships are expected to be the same, then they are really the same relationship and should not be given distinct names.

could have any structure, we shall limit isa-structures to trees, in which there is one *root* entity set (e.g., *Movies* in Fig. 2.10) that is the most general, with progressively more specialized entity sets extending below the root in a tree.

Suppose we have a tree of entity sets, connected by *isa* relationships. A single entity consists of *components* from one or more of these entity sets, as long as those components are in a subtree including the root. That is, if an entity e has a component c in entity set E, and the parent of E in the tree is F, then entity e also has a component d in F. Further, c and d must be paired in the relationship set for the *isa* relationship from E to F. The entity e has whatever attributes any of its components has, and it participates in whatever relationships any of its components participate in.

Example 2.11 : The typical movie, being neither a cartoon nor a murder-mystery, will have a component only in the root entity set *Movies* in Fig. 2.10. These entities have only the four attributes of *Movies* (and the two relationships of *Movies* — *Stars-in* and *Owns* — that are not shown in Fig. 2.10).

A cartoon that is not a murder-mystery will have two components, one in *Movies* and one in *Cartoons*. Its entity will therefore have not only the four attributes of *Movies*, but the relationship *Voices*. Likewise, a murder-mystery will have two components for its entity, one in *Movies* and one in *Murder-Mysteries* and thus will have five attributes, including *weapon*.

Finally, a movie like *Roger Rabbit*, which is both a cartoon and a murder-mystery, will have components in all three of the entity sets *Movies*, *Cartoons*, and *Murder-Mysteries*. The three components are connected into one entity by the *isa* relationships. Together, these components give the *Roger Rabbit* entity all four attributes of *Movies* plus the attribute *weapon* of entity set *Murder-Mysteries* and the relationship *Voices* of entity set *Cartoons*. □

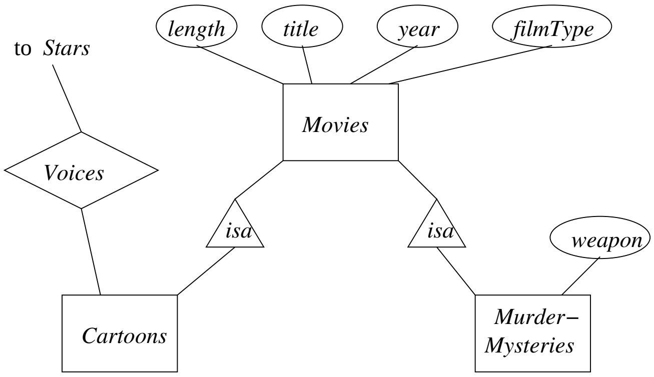

Figure 2.10: Isa relationships in an E/R diagram

2.1.12 Exercises for Section 2.1

* **Exercise 2.1.1:** Let us design a database for a bank, including information about customers and their accounts. Information about a customer includes their name, address, phone, and Social Security number. Accounts have numbers, types (e.g., savings, checking) and balances. We also need to record the customer(s) who own an account. Draw the E/R diagram for this database. Be sure to include arrows where appropriate, to indicate the multiplicity of a relationship.

Exercise 2.1.2: Modify your solution to Exercise 2.1.1 as follows:

a) Change your diagram so an account can have only one customer.

b) Further change your diagram so a customer can have only one account.

! c) Change your original diagram of Exercise 2.1.1 so that a customer can have a set of addresses (which are street-city-state triples) and a set of phones. Remember that we do not allow attributes to have nonatomic types, such as sets, in the E/R model.

! d) Further modify your diagram so that customers can have a set of addresses, and at each address there is a set of phones.

Exercise 2.1.3: Give an E/R diagram for a database recording information about teams, players, and their fans, including:

1. For each team, its name, its players, its team captain (one of its players), and the colors of its uniform.

2. For each player, his/her name.

3. For each fan, his/her name, favorite teams, favorite players, and favorite color.

Subclasses in Object-Oriented Systems

There is a significant resemblance between "isa" in the E/R model and subclasses in object-oriented languages. In a sense, "isa" relates a subclass to its superclass. However, there is also a fundamental difference between the conventional E/R view and the object-oriented approach: entities are allowed to have representatives in a tree of entity sets, while objects are assumed to exist in exactly one class or subclass.

The difference becomes apparent when we consider how the movie *Roger Rabbit* was handled in Example 2.11. In an object-oriented approach, we would need for this movie a fourth entity set, "cartoon-murder-mystery," which inherited all the attributes and relationships of *Movies*, *Cartoons*, and *Murder-Mysteries*. However, in the E/R model, the effect of this fourth subclass is obtained by putting components of the movie *Roger Rabbit* in both the *Cartoons* and *Murder-Mysteries* entity sets.

Remember that a set of colors is not a suitable attribute type for teams. How can you get around this restriction?

Exercise 2.1.4: Suppose we wish to add to the schema of Exercise 2.1.3 a relationship *Led-by* among two players and a team. The intention is that this relationship set consists of triples

$$(\text{player1, player2, team})$$

such that player 1 played on the team at a time when some other player 2 was the team captain.

a) Draw the modification to the E/R diagram.

b) Replace your ternary relationship with a new entity set and binary relationships.

! c) Are your new binary relationships the same as any of the previously existing relationships? Note that we assume the two players are different, i.e., the team captain is not self-led.

Exercise 2.1.5: Modify Exercise 2.1.3 to record for each player the history of teams on which they have played, including the start date and ending date (if they were traded) for each such team.

! **Exercise 2.1.6:** Suppose we wish to keep a genealogy. We shall have one entity set, *Person*. The information we wish to record about persons includes their name (an attribute) and the following relationships: mother, father, and children. Give an E/R diagram involving the *Person* entity set and all the relationships in which it is involved. Include relationships for mother, father, and children. Do not forget to indicate roles when an entity set is used more than once in a relationship.

! **Exercise 2.1.7:** Modify your "people" database design of Exercise 2.1.6 to include the following special types of people:

1. Females.

2. Males.

3. People who are parents.

You may wish to distinguish certain other kinds of people as well, so relationships connect appropriate subclasses of people.

Exercise 2.1.8: An alternative way to represent the information of Exercise 2.1.6 is to have a ternary relationship *Family* with the intent that a triple in the relationship set for *Family*

(person, mother, father)

is a person, their mother, and their father; all three are in the *People* entity set, of course.

* a) Draw this diagram, placing arrows on edges where appropriate.

b) Replace the ternary relationship *Family* by an entity set and binary relationships. Again place arrows to indicate the multiplicity of relationships.

Exercise 2.1.9: Design a database suitable for a university registrar. This database should include information about students, departments, professors, courses, which students are enrolled in which courses, which professors are teaching which courses, student grades, TA's for a course (TA's are students), which courses a department offers, and any other information you deem appropriate. Note that this question is more free-form than the questions above, and you need to make some decisions about multiplicities of relationships, appropriate types, and even what information needs to be represented.

! **Exercise 2.1.10:** Informally, we can say that two E/R diagrams "have the same information" if, given a real-world situation, the instances of these two diagrams that reflect this situation can be computed from one another. Consider the E/R diagram of Fig. 2.6. This four-way relationship can be decomposed into a three-way relationship and a binary relationship by taking advantage of the fact that for each movie, there is a unique studio that produces that movie. Give an E/R diagram without a four-way relationship that has the same information as Fig. 2.6.

2.2 Design Principles

We have yet to learn many of the details of the E/R model, but we have enough to begin study of the crucial issue of what constitutes a good design and what should be avoided. In this section, we offer some useful design principles.

2.2.1 Faithfulness

First and foremost, the design should be faithful to the specifications of the application. That is, entity sets and their attributes should reflect reality. You can't attach an attribute *number-of-cylinders* to *Stars*, although that attribute would make sense for an entity set *Automobiles*. Whatever relationships are asserted should make sense given what we know about the part of the real world being modeled.

Example 2.12: If we define a relationship *Stars-in* between *Stars* and *Movies*, it should be a many-many relationship. The reason is that an observation of the real world tells us that stars can appear in more than one movie, and movies can have more than one star. It is incorrect to declare the relationship `Stars-in` to be many-one in either direction or to be one-one. □

Example 2.13: On the other hand, sometimes it is less obvious what the real world requires us to do in our E/R model. Consider, for instance, entity sets *Courses* and *Instructors*, with a relationship *Teaches* between them. Is *Teaches* many-one from *Courses* to *Instructors*? The answer lies in the policy and intentions of the organization creating the database. It is possible that the school has a policy that there can be only one instructor for any course. Even if several instructors may "team-teach" a course, the school may require that exactly one of them be listed in the database as the instructor responsible for the course. In either of these cases, we would make *Teaches* a many-one relationship from *Courses* to *Instructors*.

Alternatively, the school may use teams of instructors regularly and wish its database to allow several instructors to be associated with a course. Or, the intent of the *Teaches* relationship may not be to reflect the current teacher of a course, but rather those who have ever taught the course, or those who are capable of teaching the course; we cannot tell simply from the name of the relationship. In either of these cases, it would be proper to make *Teaches* be many-many. □

2.2.2 Avoiding Redundancy

We should be careful to say everything once only. For instance, we have used a relationship *Owns* between movies and studios. We might also choose to have an attribute *studioName* of entity set *Movies*. While there is nothing illegal about doing so, it is dangerous for several reasons.

1. The two representations of the same owning-studio fact take more space, when the data is stored, than either representation alone.

2. If a movie were sold, we might change the owning studio to which it is related by relationship *Owns* but forget to change the value of its *studioName* attribute, or vice versa. Of course one could argue that one should never do such careless things, but in practice, errors are frequent, and by trying to say the same thing in two different ways, we are inviting trouble.

These problems will be described more formally in Section 3.6, and we shall also learn there some tools for redesigning database schemas so the redundancy and its attendant problems go away.

2.2.3 Simplicity Counts

Avoid introducing more elements into your design than is absolutely necessary.

Example 2.14: Suppose that instead of a relationship between *Movies* and *Studios* we postulated the existence of "movie-holdings," the ownership of a single movie. We might then create another entity set *Holdings*. A one-one relationship *Represents* could be established between each movie and the unique holding that represents the movie. A many-one relationship from *Holdings* to *Studios* completes the picture shown in Fig. 2.11.

Figure 2.11: A poor design with an unnecessary entity set

Technically, the structure of Fig. 2.11 truly represents the real world, since it is possible to go from a movie to its unique owning studio via *Holdings*. However, *Holdings* serves no useful purpose, and we are better off without it. It makes programs that use the movie-studio relationship more complicated, wastes space, and encourages errors. □

2.2.4 Choosing the Right Relationships

Entity sets can be connected in various ways by relationships. However, adding to our design every possible relationship is not often a good idea. First, it can lead to redundancy, where the connected pairs or sets of entities for one relationship can be deduced from one or more other relationships. Second, the resulting database could require much more space to store redundant elements, and modifying the database could become too complex, because one change in the data could require many changes to the stored relationships. The problems are essentially the same as those discussed in Section 2.2.2, although the cause of the problem is different from the problems we discussed there.

We shall illustrate the problem and what to do about it with two examples. In the first example, several relationships could represent the same information; in the second, one relationship could be deduced from several others.

Example 2.15: Let us review Fig. 2.7, where we connected movies, stars, and studios with a three-way relationship *Contracts*. We omitted from that figure the two binary relationships *Stars-in* and *Owns* from Fig. 2.2. Do we also need these relationships, between *Movies* and *Stars*, and between *Movies* and *Studios*, respectively? The answer is: "we don't know; it depends on our assumptions regarding the three relationships in question."

It might be possible to deduce the relationship *Stars-in* from *Contracts*. If a star can appear in a movie only if there is a contract involving that star, that movie, and the owning studio for the movie, then there truly is no need for relationship *Stars-in*. We could figure out all the star-movie pairs by looking at the star-movie-studio triples in the relationship set for *Contracts* and taking only the star and movie components. However, if a star can work on a movie without there being a contract — or what is more likely, without there being a contract that we know about in our database — then there could be star-movie pairs in *Stars-in* that are not part of star-movie-studio triples in *Contracts*. In that case, we need to retain the *Stars-in* relationship.

A similar observation applies to relationship *Owns*. If for every movie, there is at least one contract involving that movie, its owning studio, and some star for that movie, then we can dispense with *Owns*. However, if there is the possibility that a studio owns a movie, yet has no stars under contract for that movie, or no such contract is known to our database, then we must retain *Owns*.

In summary, we cannot tell you whether a given relationship will be redundant. You must find out from those who wish the database created what to expect. Only then can you make a rational decision about whether or not to include relationships such as *Stars-in* or *Owns*. □

Example 2.16: Now, consider Fig. 2.2 again. In this diagram, there is no relationship between stars and studios. Yet we can use the two relationships *Stars-in* and *Owns* to build a connection by the process of composing those two relationships. That is, a star is connected to some movies by *Stars-in*, and those movies are connected to studios by *Owns*. Thus, we could say that a star is connected to the studios that own movies in which the star has appeared.

Would it make sense to have a relationship *Works-for*, as suggested in Fig. 2.12, between *Stars* and *Studios* too? Again, we cannot tell without knowing more. First, what would the meaning of this relationship be? If it is to mean "the star appeared in at least one movie of this studio," then probably there is no good reason to include it in the diagram. We could deduce this information from *Stars-in* and *Owns* instead.

However, it is conceivable that we have other information about stars working for studios that is not entailed by the connection through a movie. In that

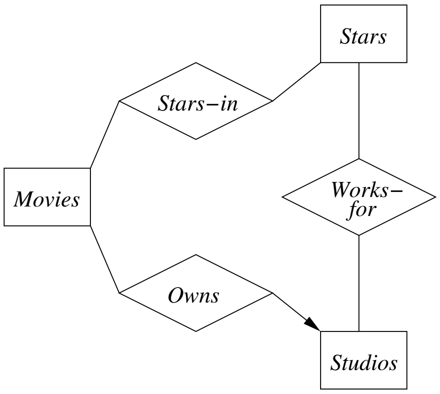

Figure 2.12: Adding a relationship between *Stars* and *Studios*

case, a relationship connecting stars directly to studios might be useful and would not be redundant. Alternatively, we might use a relationship between stars and studios to mean something entirely different. For example, it might represent the fact that the star is under contract to the studio, in a manner unrelated to any movie. As we suggested in Example 2.7, it is possible for a star to be under contract to one studio and yet work on a movie owned by another studio. In this case, the information found in the new *Works-for* relation would be independent of the *Stars-in* and *Owns* relationships, and would surely be nonredundant. □

2.2.5 Picking the Right Kind of Element

Sometimes we have options regarding the type of design element used to represent a real-world concept. Many of these choices are between using attributes and using entity set/relationship combinations. In general, an attribute is simpler to implement than either an entity set or a relationship. However, making everything an attribute will usually get us into trouble.

Example 2.17: Let us consider a specific problem. In Fig. 2.2, were we wise to make studios an entity set? Should we instead have made the name and address of the studio be attributes of movies and eliminated the *Studio* entity set? One problem with doing so is that we repeat the address of the studio for each movie. This situation is another instance of redundancy, similar to those seen in Sections 2.2.2 and 2.2.4. In addition to the disadvantages of redundancy discussed there, we also face the risk that, should we not have any movies owned by a given studio, we lose the studio's address.

On the other hand, if we did not record addresses of studios, then there is no harm in making the studio name an attribute of movies. We do not have redundancy due to repeating addresses. The fact that we have to say the name of a studio like `Disney` for each movie owned by Disney is not true redundancy, since we must represent the owner of each movie somehow, and saying the name is a reasonable way to do so. □

We can abstract what we have observed in Example 2.17 to give the conditions under which we prefer to use an attribute instead of an entity set. Suppose E is an entity set. Here are conditions that E must obey, in order for us to replace E by an attribute or attributes of several other entity sets.

1. All relationships in which E is involved must have arrows entering E. That is, E must be the "one" in many-one relationships, or its generalization for the case of multiway relationships.

2. The attributes for E must collectively identify an entity. Typically, there will be only one attribute, in which case this condition is surely met. However, if there are several attributes, then no attribute must depend on the other attributes, the way *address* depends on *name* for *Studios*.

3. No relationship involves E more than once.

If these conditions are met, then we can replace entity set E as follows:

a) If there is a many-one relationship R from some entity set F to E, then remove R and make the attributes of E be attributes of F, suitably renamed if they conflict with attribute names for F. In effect, each F-entity takes, as attributes, the name of the unique, related E-entity,[2] as movie objects could take their studio name as an attribute, should we dispense with studio addresses.

b) If there is a multiway relationship R with an arrow to E, make the attributes of E be attributes of R and delete the arc from R to E. An example of transformation is replacing Fig. 2.8, where we had introduced a new entity set *Salaries*, with a number as its lone attribute, by its original diagram, in Fig. 2.7.

Example 2.18: Let us consider a point where there is a tradeoff between using a multiway relationship and using a connecting entity set with several binary relationships. We saw a four-way relationship *Contracts* among a star, a movie, and two studios in Fig. 2.6. In Fig. 2.9, we mechanically converted it to an entity set *Contracts*. Does it matter which we choose?

As the problem was stated, either is appropriate. However, should we change the problem just slightly, then we are almost forced to choose a connecting entity set. Let us suppose that contracts involve one star, one movie, but any set of studios. This situation is more complex than the one in Fig. 2.6, where we had two studios playing two roles. In this case, we can have any number of

[2] In a situation where an F-entity is not related to any E-entity, the new attributes of F would be given special "null" values to indicate the absence of a related E-entity. A similar arrangement would be used for the new attributes of R in case (b).

studios involved, perhaps one to do production, one for special effects, one for distribution, and so on. Thus, we cannot assign roles for studios.

It appears that a relationship set for the relationship *Contracts* must contain triples of the form

(star, movie, set-of-studios)

and the relationship *Contracts* itself involves not only the usual *Stars* and *Movies* entity sets, but a new entity set whose entities are *sets of* studios. While this approach is unpreventable, it seems unnatural to think of sets of studios as basic entities, and we do not recommend it.

A better approach is to think of contracts as an entity set. As in Fig. 2.9, a contract entity connects a star, a movie and a set of studios, but now there must be no limit on the number of studios. Thus, the relationship between contracts and studios is many-many, rather than many-one as it would be if contracts were a true "connecting" entity set. Figure 2.13 sketches the E/R diagram. Note that a contract is associated with a single star and a single movie, but any number of studios. □

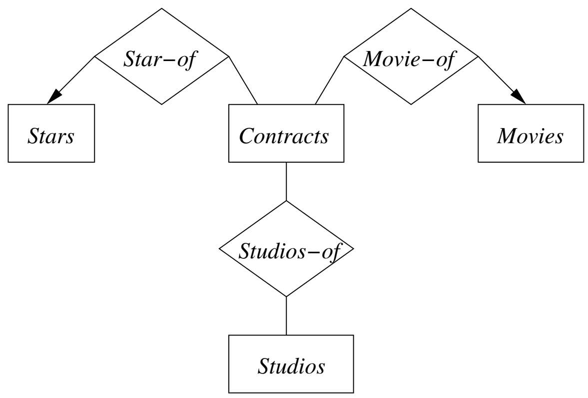

Figure 2.13: Contracts connecting a star, a movie, and a set of studios

2.2.6 Exercises for Section 2.2

* **Exercise 2.2.1:** In Fig. 2.14 is an E/R diagram for a bank database involving customers and accounts. Since customers may have several accounts, and accounts may be held jointly by several customers, we associate with each customer an "account set," and accounts are members of one or more account sets. Assuming the meaning of the various relationships and attributes are as expected given their names, criticize the design. What design rules are violated? Why? What modifications would you suggest?

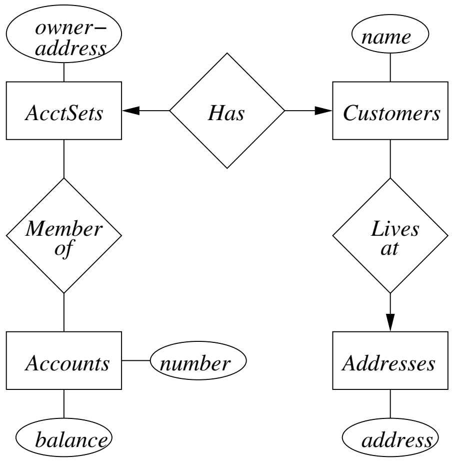

Figure 2.14: A poor design for a bank database

* **Exercise 2.2.2:** Under what circumstances (regarding the unseen attributes of *Studios* and *Presidents* would you recommend combining the two entity sets and relationship in Fig. 2.3 into a single entity set and attributes?

Exercise 2.2.3: Suppose we delete the attribute *address* from *Studios* in Fig. 2.7. Show how we could then replace an entity set by an attribute. Where would that attribute appear?

Exercise 2.2.4: Give choices of attributes for the following entity sets in Fig. 2.13 that will allow the entity set to be replaced by an attribute:

a) *Stars*.

b) *Movies*.

! c) *Studios*.

!! **Exercise 2.2.5:** In this and following exercises we shall consider two design options in the E/R model for describing births. At a birth, there is one baby (twins would be represented by two births), one mother, any number of nurses, and any number of doctors. Suppose, therefore, that we have entity sets *Babies*, *Mothers*, *Nurses*, and *Doctors*. Suppose we also use a relationship *Births*, which connects these four entity sets, as suggested in Fig. 2.15. Note that a tuple of the relationship set for *Births* has the form

(baby, mother, nurse, doctor)

If there is more than one nurse and/or doctor attending a birth, then there will be several tuples with the same baby and mother, one for each combination of nurse and doctor.

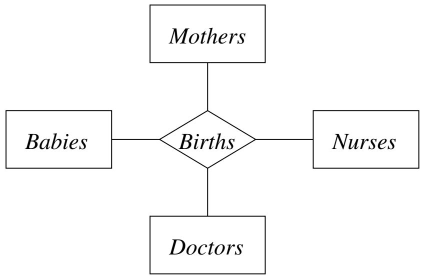

Figure 2.15: Representing births by a multiway relationship

There are certain assumptions that we might wish to incorporate into our design. For each, tell how to add arrows or other elements to the E/R diagram in order to express the assumption.

a) For every baby, there is a unique mother.

b) For every combination of a baby, nurse, and doctor, there is a unique mother.

c) For every combination of a baby and a mother there is a unique doctor.

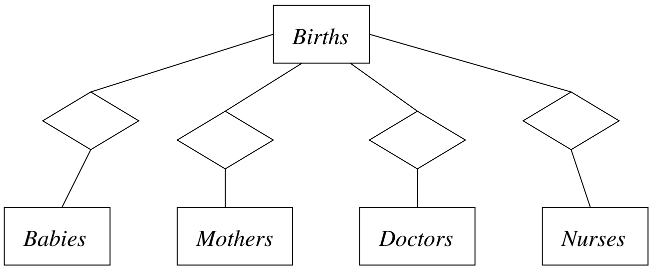

Figure 2.16: Representing births by an entity set

! **Exercise 2.2.6:** Another approach to the problem of Exercise 2.2.5 is to connect the four entity sets *Babies*, *Mothers*, *Nurses*, and *Doctors* by an entity set *Births*, with four relationships, one between *Births* and each of the other entity sets, as suggested in Fig. 2.16. Use arrows (indicating that certain of these relationships are many-one) to represent the following conditions:

a) Every baby is the result of a unique birth, and every birth is of a unique baby.

b) In addition to (a), every baby has a unique mother.

c) In addition to (a) and (b), for every birth there is a unique doctor.

In each case, what design flaws do you see?

!! **Exercise 2.2.7:** Suppose we change our viewpoint to allow a birth to involve more than one baby born to one mother. How would you represent the fact that every baby still has a unique mother using the approaches of Exercises 2.2.5 and 2.2.6?

2.3 The Modeling of Constraints

We have seen so far how to model a slice of the real world using entity sets and relationships. However, there are some other important aspects of the real world that we cannot model with the tools seen so far. This additional information often takes the form of *constraints* on the data that go beyond the structural and type constraints imposed by the definitions of entity sets, attributes, and relationships.

2.3.1 Classification of Constraints

The following is a rough classification of commonly used constraints. We shall not cover all of these constraint types here. Additional material on constraints is found in Section 5.5 in the context of relational algebra and in Chapter 7 in the context of SQL programming.

1. *Keys* are attributes or sets of attributes that uniquely identify an entity within its entity set. No two entities may agree in their values for all of the attributes that constitute a key. It *is* permissible, however, for two entities to agree on some, but not all, of the key attributes.

2. *Single-value constraints* are requirements that the value in a certain context be unique. Keys are a major source of single-value constraints, since they require that each entity in an entity set has unique value(s) for the key attribute(s). However, there are other sources of single-value constraints, such as many-one relationships.

3. *Referential integrity constraints* are requirements that a value referred to by some object actually exists in the database. Referential integrity is analogous to a prohibition against dangling pointers, or other kinds of dangling references, in conventional programs.

4. *Domain constraints* require that the value of an attribute must be drawn from a specific set of values or lie within a specific range.

5. *General constraints* are arbitrary assertions that are required to hold in the database. For example, we might wish to require that no more than ten stars be listed for any one movie. We shall see general constraint-expression languages in Sections 5.5 and 7.4.

There are several ways these constraints are important. They tell us something about the structure of those aspects of the real world that we are modeling. For example, keys allow the user to identify entities without confusion. If we know that attribute *name* is a key for entity set *Studios*, then when we refer to a studio entity by its name we know we are referring to a unique entity. In addition, knowing a unique value exists saves space and time, since storing a single value is easier than storing a set, even when that set has exactly one member.[3] Referential integrity and keys also support certain storage structures that allow faster access to data, as we shall discuss in Chapter 13.

2.3.2 Keys in the E/R Model

A *key* for an entity set E is a set K of one or more attributes such that, given any two distinct entities e_1 and e_2 in E, e_1 and e_2 cannot have identical values for each of the attributes in the key K. If K consists of more than one attribute, then it is possible for e_1 and e_2 to agree in some of these attributes, but never in all attributes. Three useful points to remember are:

- A key can consist of more than one attribute; an illustration appears in Example 2.19.
- There can also be more than one possible key for an entity set, as we shall see in Example 2.20. However, it is customary to pick one key as the "primary key," and to act as if that were the only key.
- When an entity set is involved in an isa-hierarchy, we require that the root entity set have all the attributes needed for a key, and that the key for each entity is found from its component in the root entity set, regardless of how many entity sets in the hierarchy have components for the entity in question.

Example 2.19: Let us consider the entity set *Movies* from Example 2.1. One might first assume that the attribute *title* by itself is a key. However, there are several titles that have been used for two or even more movies, for example, *King Kong*. Thus, it would be unwise to declare that *title* by itself is a key. If we did so, then we would not be able to include information about both *King Kong* movies in our database.

A better choice would be to take the set of two attributes *title* and *year* as a key. We still run the risk that there are two movies made in the same year with the same title (and thus both could not be stored in our database), but that is unlikely.

For the other two entity sets, *Stars* and *Studios*, introduced in Example 2.1, we must again think carefully about what can serve as a key. For studios, it is reasonable to assume that there would not be two movie studios with the same

[3] In analogy, note that in a C program it is simpler to represent an integer than it is to represent a linked list of integers, even when that list contains only one integer.

Constraints Are Part of the Schema

We could look at the database as it exists at a certain time and decide erroneously that an attribute forms a key because no two entities have identical values for this attribute. For example, as we create our movie database we might not enter two movies with the same title for some time. Thus, it might look as if *title* were a key for entity set *Movies*. However, if we decided on the basis of this preliminary evidence that *title* is a key, and we designed a storage structure for our database that assumed *title* is a key, then we might find ourselves unable to enter a second *King Kong* movie into the database.

Thus, key constraints, and constraints in general, are part of the database schema. They are declared by the database designer along with the structural design (e.g., entities and relationships). Once a constraint is declared, insertions or modifications to the database that violate the constraint are disallowed.

Hence, although a particular instance of the database may satisfy certain constraints, the only "true" constraints are those identified by the designer as holding for all instances of the database that correctly model the real-world. These are the constraints that may be assumed by users and by the structures used to store the database.

name, so we shall take *name* to be a key for entity set *Studios*. However, it is less clear that stars are uniquely identified by their name. Surely name does not distinguish among people in general. However, since stars have traditionally chosen "stage names" at will, we might hope to find that *name* serves as a key for *Stars* too. If not, we might choose the pair of attributes *name* and *address* as a key, which would be satisfactory unless there were two stars with the same name living at the same address. □

Example 2.20: Our experience in Example 2.19 might lead us to believe that it is difficult to find keys or to be sure that a set of attributes forms a key. In practice the matter is usually much simpler. In the real-world situations commonly modeled by databases, people often go out of their way to create keys for entity sets. For example, companies generally assign employee ID's to all employees, and these ID's are carefully chosen to be unique numbers. One purpose of these ID's is to make sure that in the company database each employee can be distinguished from all others, even if there are several employees with the same name. Thus, the employee-ID attribute can serve as a key for employees in the database.

In US corporations, it is normal for every employee to also have a Social Security number. If the database has an attribute that is the Social Security number, then this attribute can also serve as a key for employees. Note that there is nothing wrong with there being several choices of key for an entity set, as there would be for employees having both employee ID's and Social Security numbers.

The idea of creating an attribute whose purpose is to serve as a key is quite widespread. In addition to employee ID's, we find student ID's to distinguish students in a university. We find drivers' license numbers and automobile registration numbers to distinguish drivers and automobiles, respectively, in the Department of Motor Vehicles. The reader can undoubtedly find more examples of attributes created for the primary purpose of serving as keys. □

2.3.3 Representing Keys in the E/R Model

In our E/R diagram notation, we underline the attributes belonging to a key for an entity set. For example, Fig. 2.17 reproduces our E/R diagram for movies, stars, and studios from Fig. 2.2, but with key attributes underlined. Attribute *name* is the key for *Stars*. Likewise, *Studios* has a key consisting of only its own attribute *name*. These choices are consistent with the discussion in Example 2.19.

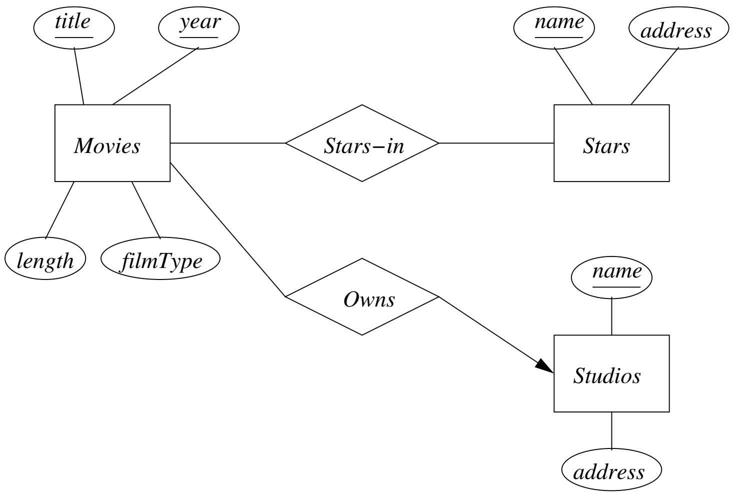

Figure 2.17: E/R diagram; keys are indicated by underlines

The attributes *title* and *year* together form the key for *Movies*, as we discussed in Example 2.19. Note that when several attributes are underlined, as in Fig. 2.17, then they are each members of the key. There is no notation for representing the situation where there are several keys for an entity set; we underline only the primary key. You should also be aware that in some unusual situations, the attributes forming the key for an entity set do not all belong to the entity set itself. We shall defer this matter, called "weak entity sets," until Section 2.4.

2.3.4 Single-Value Constraints

Often, an important property of a database design is that there is at most one value playing a particular role. For example, we assume that a movie entity has a unique title, year, length, and film type, and that a movie is owned by a unique studio.

There are several ways in which single-value constraints are expressed in the E/R model.

1. Each attribute of an entity set has a single value. Sometimes it is permissible for an attribute's value to be missing for some entities, in which case we have to invent a "null value" to serve as the value of that attribute. For example, we might suppose that there are some movies in our database for which the length is not known. We could use a value such as -1 for the length of a movie whose true length is unknown. On the other hand, we would not want the key attributes *title* or *year* to be null for any movie entity. A requirement that a certain attribute not have a null value does not have any special representation in the E/R model. We could place a notation beside the attribute stating this requirement if we wished.

2. A relationship R that is many-one from entity set E to entity set F implies a single-value constraint. That is, for each entity e in E, there is at most one associated entity f in F. More generally, if R is a multiway relationship, then each arrow out of R indicates a single value constraint. Specifically, if there is an arrow from R to entity set E, then there is at most one entity of set E associated with a choice of entities from each of the other related entity sets.

2.3.5 Referential Integrity

While single-value constraints assert that at most one value exists in a given role, a *referential integrity constraint* asserts that exactly one value exists in that role. We could see a constraint that an attribute have a non-null, single value as a kind of referential integrity requirement, but "referential integrity" is more commonly used to refer to relationships among entity sets.

Let us consider the many-one relationship *Owns* from *Movies* to *Studios* in Fig. 2.2. The many-one requirement simply says that no movie can be owned by more than one studio. It does *not* say that a movie must surely be owned by a studio, or that, even if it is owned by some studio, that the studio must be present in the *Studios* entity set, as stored in our database.

A referential integrity constraint on relationship *Owns* would require that for each movie, the owning studio (the entity "referenced" by the relationship for this movie) must exist in our database. There are several ways this constraint could be enforced.

1. We could forbid the deletion of a referenced entity (a studio in our example). That is, we could not delete a studio from the database unless it did not own any movies.

2. We could require that if a referenced entity is deleted, then all entities that reference it are deleted as well. In our example, this approach would require that if we delete a studio, we also delete from the database all movies owned by that studio.

In addition to one of these policies about deletion, we require that when a movie entity is inserted into the database, it is given an existing studio entity to which it is connected by relationship *Owns*. Further, if the value of that relationship changes, then the new value must also be an existing *Studios* entity. Enforcing these policies to assure referential integrity of a relationship is a matter for the implementation of the database, and we shall not discuss the details here.

2.3.6 Referential Integrity in E/R Diagrams

We can extend the arrow notation in E/R diagrams to indicate whether a relationship is expected to support referential integrity in one or more directions. Suppose R is a relationship from entity set E to entity set F. We shall use a rounded arrowhead pointing to F to indicate not only that the relationship is many-one or one-one from E to F, but that the entity of set F related to a given entity of set E is required to exist. The same idea applies when R is a relationship among more than two entity sets.

Example 2.21 : Figure 2.18 shows some appropriate referential integrity constraints among the entity sets *Movies*, *Studios*, and *Presidents*. These entity sets and relationships were first introduced in Figs. 2.2 and 2.3. We see a rounded arrow entering *Studios* from relationship *Owns*. That arrow expresses the referential integrity constraint that every movie must be owned by one studio, and this studio is present in the *Studios* entity set.

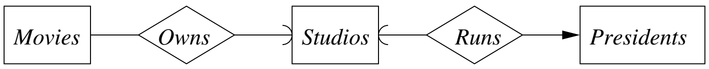

Figure 2.18: E/R diagram showing referential integrity constraints

Similarly, we see a rounded arrow entering *Studios* from *Runs*. That arrow expresses the referential integrity constraint that every president runs a studio that exists in the *Studios* entity set.

Note that the arrow to *Presidents* from *Runs* remains a pointed arrow. That choice reflects a reasonable assumption about the relationship between studios and their presidents. If a studio ceases to exist, its president can no longer be called a (studio) president, so we would expect the president of the studio to be deleted from the entity set *Presidents*. Hence there is a rounded arrow to *Studios*. On the other hand, if a president were deleted from the database, the studio would continue to exist. Thus, we place an ordinary, pointed arrow to *Presidents*, indicating that each studio has at most one president, but might have no president at some time. □

2.3.7 Other Kinds of Constraints

As mentioned at the beginning of this section, there are other kinds of constraints one could wish to enforce in a database. We shall only touch briefly on these here, with the meat of the subject appearing in Chapter 7.

Domain constraints restrict the value of an attribute to be in a limited set. A simple example would be declaring the type of an attribute. A stronger domain constraint would be to declare an enumerated type for an attribute or a range of values, e.g., the *length* attribute for a movie must be an integer in the range 0 to 240. There is no specific notation for domain constraints in the E/R model, but you may place a notation stating a desired constraint next to the attribute, if you wish.

There are also more general kinds of constraints that do not fall into any of the categories mentioned in this section. For example, we could choose to place a constraint on the degree of a relationship, such as that a movie entity cannot be connected by relationship `stars` to more than 10 star entities. In the E/R model, we can attach a bounding number to the edges that connect a relationship to an entity set, indicating limits on the number of entities that can be connected to any one entity of the related entity set.

Figure 2.19: Representing a constraint on the number of stars per movie

Example 2.22: Figure 2.19 shows how we can represent the constraint that no movie has more than 10 stars in the E/R model. As another example, we can think of the arrow as a synonym for the constraint "≤ 1," and we can think of the rounded arrow of Fig. 2.18 as standing for the constraint "$= 1$." □

2.3.8 Exercises for Section 2.3

Exercise 2.3.1: For your E/R diagrams of:

* a) Exercise 2.1.1.

b) Exercise 2.1.3.

c) Exercise 2.1.6.

(*i*) Select and specify keys, and (*ii*) Indicate appropriate referential integrity constraints.

! **Exercise 2.3.2:** We may think of relationships in the E/R model as having keys, just as entity sets do. Let R be a relationship among the entity sets $E_1, E_2, \ldots, E_n$. Then a *key* for R is a set K of attributes chosen from the attributes of $E_1, E_2, \ldots, E_n$ such that if $(e_1, e_2, \ldots, e_n)$ and $(f_1, f_2, \ldots, f_n)$ are two different tuples in the relationship set for R, then it is not possible that these tuples agree in all the attributes of K. Now, suppose $n = 2$; that is, R is a binary relationship. Also, for each i, let K_i be a set of attributes that is a key for entity set E_i. In terms of E_1 and E_2, give a smallest possible key for R under the assumption that:

a) R is many-many.

* b) R is many-one from E_1 to E_2.

c) R is many-one from E_2 to E_1.

d) R is one-one.

!! **Exercise 2.3.3:** Consider again the problem of Exercise 2.3.2, but with n allowed to be any number, not just 2. Using only the information about which arcs from R to the E_i's have arrows, show how to find a smallest possible key K for R in terms of the K_i's.

! **Exercise 2.3.4:** Give examples (other than those of Example 2.20) from real life of attributes created for the primary purpose of being keys.

2.4 Weak Entity Sets

There is an occasional condition in which an entity set's key is composed of attributes some or all of which belong to another entity set. Such an entity set is called a *weak entity set.*

2.4.1 Causes of Weak Entity Sets

There are two principal sources of weak entity sets. First, sometimes entity sets fall into a hierarchy based on classifications unrelated to the "isa hierarchy" of Section 2.1.11. If entities of set E are subunits of entities in set F, then it is possible that the names of E entities are not unique until we take into account the name of the F entity to which the E entity is subordinate. Several examples will illustrate the problem.

Example 2.23: A movie studio might have several film crews. The crews might be designated by a given studio as crew 1, crew 2, and so on. However, other studios might use the same designations for crews, so the attribute *number* is not a key for crews. Rather, to name a crew uniquely, we need to give both the name of the studio to which it belongs and the number of the crew. The situation is suggested by Fig. 2.20. The key for weak entity set *Crews* is its own *number* attribute and the *name* attribute of the unique studio to which the crew is related by the many-one *Unit-of* relationship.[4] □

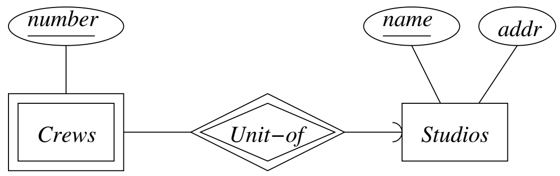

Figure 2.20: A weak entity set for crews, and its connections

Example 2.24: A species is designated by its genus and species names. For example, humans are of the species *Homo sapiens*; *Homo* is the genus name and *sapiens* the species name. In general, a genus consists of several species, each of which has a name beginning with the genus name and continuing with the species name. Unfortunately, species names, by themselves, are not unique. Two or more genera may have species with the same species name. Thus, to designate a species uniquely we need both the species name and the name of the genus to which the species is related by the *Belongs-to* relationship, as suggested in Fig. 2.21. *Species* is a weak entity set whose key comes partially from its genus. □

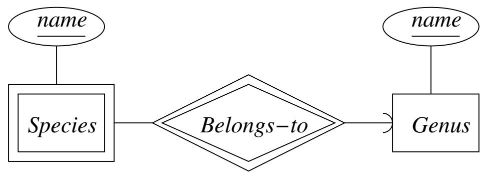

Figure 2.21: Another weak entity set, for species

The second common source of weak entity sets is the connecting entity sets that we introduced in Section 2.1.10 as a way to eliminate a multiway relationship.[5] These entity sets often have no attributes of their own. Their

[4] The double diamond and double rectangle will be explained in Section 2.4.3.

[5] Remember that there is no particular requirement in the E/R model that multiway relationships be eliminated, although this requirement exists in some other database design models.

key is formed from the attributes that are the key attributes for the entity sets they connect.

Example 2.25: In Fig. 2.22 we see a connecting entity set *Contracts* that replaces the ternary relationship *Contracts* of Example 2.5. *Contracts* has an attribute *salary*, but this attribute does not contribute to the key. Rather, the key for a contract consists of the name of the studio and the star involved, plus the title and year of the movie involved. □

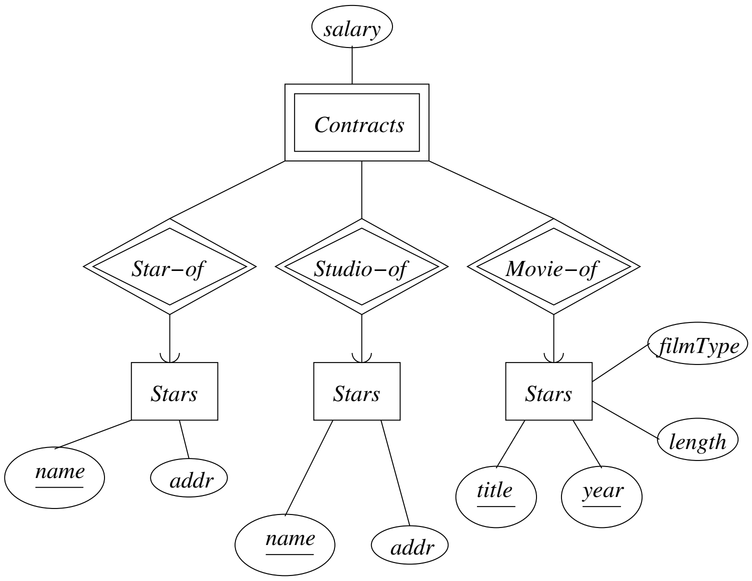

Figure 2.22: Connecting entity sets are weak

2.4.2 Requirements for Weak Entity Sets

We cannot obtain key attributes for a weak entity set indiscriminately. Rather, if E is a weak entity set then its key consists of:

1. Zero or more of its own attributes, and

2. Key attributes from entity sets that are reached by certain many-one relationships from E to other entity sets. These many-one relationships are called *supporting relationships* for E.

In order for R, a many-one relationship from E to some entity set F, to be a supporting relationship for E, the following conditions must be obeyed:

a) R must be a binary, many-one relationship[6] from E to F.

[6] Remember that a one-one relationship is a special case of a many-one relationship. When we say a relationship must be many-one, we always include one-one relationships as well.

b) R must have referential integrity from E to F. That is, for every E-entity, the F-entity related to it by R must actually exist in the database. Put another way, a rounded arrow from R to F must be justified.

c) The attributes that F supplies for the key of E must be key attributes of F.

d) However, if F is itself weak, then some or all of the key attributes of F supplied to E will be key attributes of one or more entity sets G to which F is connected by a supporting relationship. Recursively, if G is weak, some key attributes of G will be supplied from elsewhere, and so on.

e) If there are several different supporting relationships from E to F, then each relationship is used to supply a copy of the key attributes of F to help form the key of E. Note that an entity e from E may be related to different entities in F through different supporting relationships from E. Thus, the keys of several different entities from F may appear in the key values identifying a particular entity e from E.

The intuitive reason why these conditions are needed is as follows. Consider an entity in a weak entity set, say a crew in Example 2.23. Each crew is unique, abstractly. In principle we can tell one crew from another, even if they have the same number but belong to different studios. It is only the data about crews that makes it hard to distinguish crews, because the number alone is not sufficient. The only way we can associate additional information with a crew is if there is some deterministic process leading to additional values that make the designation of a crew unique. But the only unique values associated with an abstract crew entity are:

1. Values of attributes of the *Crews* entity set, and

2. Values obtained by following a relationship from a crew entity to a unique entity of some other entity set, where that other entity has a unique associated value of some kind. That is, the relationship followed must be many-one (or one-one as a special case) to the other entity set F, and the associated value must be part of a key for F.

2.4.3 Weak Entity Set Notation

We shall adopt the following conventions to indicate that an entity set is weak and to declare its key attributes.

1. If an entity set is weak, it will be shown as a rectangle with a double border. Examples of this convention are *Crews* in Fig. 2.20 and *Contracts* in Fig. 2.22.

2. Its supporting many-one relationships will be shown as diamonds with a double border. Examples of this convention are *Unit-of* in Fig. 2.20 and all three relationships in Fig. 2.22.

3. If an entity set supplies any attributes for its own key, then those attributes will be underlined. An example is in Fig. 2.20, where the number of a crew participates in its own key, although it is not the complete key for *Crews*.

We can summarize these conventions with the following rule:

- Whenever we use an entity set E with a double border, it is weak. E's attributes that are underlined, if any, plus the key attributes of those entity sets to which E is connected by many-one relationships with a double border, must be unique for the entities of E.

We should remember that the double-diamond is used only for supporting relationships. It is possible for there to be many-one relationships from a weak entity set that are not supporting relationships, and therefore do not get a double diamond.

Example 2.26: In Fig. 2.22, the relationship *Studio-of* need not be a supporting relationship for *Contracts*. The reason is that each movie has a unique owning studio, determined by the (not shown) many-one relationship from *Movies* to *Studios*. Thus, if we are told the name of a star and a movie, there is at most one contract with any studio for the work of that star in that movie. In terms of our notation, it would be appropriate to use an ordinary single diamond, rather than the double diamond, for *Studio-of* in Fig. 2.22. □

2.4.4 Exercises for Section 2.4

* **Exercise 2.4.1:** One way to represent students and the grades they get in courses is to use entity sets corresponding to students, to courses, and to "enrollments." Enrollment entities form a "connecting" entity set between students and courses and can be used to represent not only the fact that a student is taking a certain course, but the grade of the student in the course. Draw an E/R diagram for this situation, indicating weak entity sets and the keys for the entity sets. Is the grade part of the key for enrollments?

Exercise 2.4.2: Modify your solution to Exercise 2.4.1 so that we can record grades of the student for each of several assignments within a course. Again, indicate weak entity sets and keys.

Exercise 2.4.3: For your E/R diagrams of Exercise 2.2.6(a)–(c), indicate weak entity sets, supporting relationships, and keys.

Exercise 2.4.4: Draw E/R diagrams for the following situations involving weak entity sets. In each case indicate keys for entity sets.

a) Entity sets *Courses* and *Departments*. A course is given by a unique department, but its only attribute is its number. Different departments can offer courses with the same number. Each department has a unique name.

*! b) Entity sets *Leagues*, *Teams*, and *Players*. League names are unique. No league has two teams with the same name. No team has two players with the same number. However, there can be players with the same number on different teams, and there can be teams with the same name in different leagues.

2.5 Summary of Chapter 2

- *The Entity/Relationship Model*: In the E/R model we describe entity sets, relationships among entity sets, and attributes of entity sets and relationships. Members of entity sets are called entities.

- *Entity/Relationship Diagrams*: We use rectangles, diamonds, and ovals to draw entity sets, relationships, and attributes, respectively.

- *Multiplicity of Relationships*: Binary relationships can be one-one, many-one, or many-many. In a one-one relationship, an entity of either set can be associated with at most one entity of the other set. In a many-one relationship, each entity of the "many" side is associated with at most one entity of the other side. Many-many relationships place no restriction on multiplicity.

- *Keys*: A set of attributes that uniquely determines an entity in a given entity set is a key for that entity set.

- *Good Design*: Designing databases effectively requires that we represent the real world faithfully, that we select appropriate elements (e.g., relationships, attributes), and that we avoid redundancy — saying the same thing twice or saying something in an indirect or overly complex manner.

- *Referential Integrity*: A requirement that an entity be connected, through a given relationship, to an entity of some other entity set, and that the latter entity exists in the database, is called a referential integrity constraint.

- *Subclasses*: The E/R model uses a special relationship *isa* to represent the fact that one entity set is a special case of another. Entity sets may be connected in a hierarchy with each child node a special case of its parent. Entities may have components belonging to any subtree of the hierarchy, as long as the subtree includes the root.

- *Weak Entity Sets*: An occasional complication that arises in the E/R model is a weak entity set that requires attributes of some related entity set(s) to identify its own entities. A special notation involving diamonds and rectangles with double borders is used to distinguish weak entity sets.

2.6 References for Chapter 2

The original paper on the Entity/Relationship model is [2]. Two modern books on the subject of E/R design are [1] and [3].

1. Batini, Carlo., S. Ceri, and S. B. Navathe, and Carol Batini, *Conceptual Database Design: an Entity/Relationship Approach*, Addison-Wesley, Reading MA, 1991.

2. Chen, P. P., "The entity-relationship model: toward a unified view of data," *ACM Trans. on Database Systems* **1**:1, pp. 9–36, 1976.

3. Thalheim, B., "Fundamentals of Entity-Relationship Modeling," Springer-Verlag, Berlin, 2000.